Dokumentation Architektur verteilter Anwendungen

Jan Zipfler (3553248)

Dokumentation Architektur verteilter Anwendungen

Jan Zipfler (3553248)

Table of Contents

List of Figures

Chapter 1. Programmiersprache

Für dieses Projekt wurde die Programmiersprache GO verwendet. Dabei handelt es sich um eine von Google entwickelte Programmiersprache, die zu Beginn in einem der 20% Projekte entwickelt wurde. Hauptverantworklich für die Sprache sind die Entwickler:

- Robert Griesemer
- Rob Pike
  - Arbeitete an der Programmiersprache Limbo mit, welche sich auf der Erstellung von verteilten Systemen konzentrierte.
- Ken Thompson
  - Unteranderem bekannt für das Mitwirken an der Erstellung der Sprache C.

GO brüstet sich damit, für verteilte Anwendungen und vorallem Webservices entwicklet worden zu sein, wodurch es in meinen Augen eine gute Wahl für dieses Fach zu sein schien. Außerdem erleichtert die Sprache zu einen die parallele Ausführung von Code und zum anderen durch das Prinzip von Channels die Interprozesskommunikation.

Als Entwicklungsumgebung kann ich die LiteIDE empfehlen, da andere Entwicklungsumgebungen einen geringeren Umfang bietet und sich schwieriger bedienen lassen. Des weiteren hat mich das Werk von *Mark Summerfield* (Programming in Go) begleitet und mit den dort vorhanden Beispielen des öfteren weitergeholfen.

1.1. Installation der benötigten Pakete

Um alle Pakete zur Übersetzung meiner Programme zu bekommen können die von Go zur Verfügung gestellten Tools genutzt werden. Da mein Quellcode auf GitHub öffentlich zur Verfügung steht, kann mein Projekt mit dem folgendem Befehl heruntergeladen werden:

```
go get -u github.com/jzipfler/HTW-AVA/{avaStarter,avaStarter2,exercise3/fileManager,exer
```

Sollte dies fehlschlagen kann das Repository mithilfe von git wie folgt geklont werden:

```
git clone github.com/jzipfler/HTW-AVA/ ${GOPATH}/src/github.com/jzipfler/HTW-AVA
```

Nachdem die Dateien auf der Festplatte sind, können die Pakete entweder über die Makefiles in den *example* Ordnern, oder über die folgenden Befehle übersetzt werden:

```
git build -v github.com/jzipfler/HTW-AVA/{avaStarter,avaStarter2,exercise3/fileManager,e
```

Dabei muss beachtet werden, dass dieses Projekt eine Abhängigkeit zu Protobuf aufweist, welches einmal in der Version von googlecode und von github benötigt wird.

Chapter 2. Exercise 1

2.1. Erläuterung der Idee

Da ich bereits im letzten Jahr an der Vorlesung teilgenommen hatte hatte ich bei der thematischen Bearbeitung von Aufgabe 1 keine Probleme. Schwieriger war es jedoch, wie auch schon im Jahr zuvor, sich in die neue Programmiersprache so einzuarbeiten, dass ein zum einen schönes und zum anderes gutes Ergebnis heraus kommt.

Dabei habe ich dieses Jahr, zumindest zu Beginn, darauf Wert gelegt, dass die Anwendung die bereitgestellten Möglichkeiten von Go nutzt. Darunter gehörte zum einen, dass mir durch die Consistec näher gebrachte Prinzip der testgetriebenen Entwicklung.

Ansonsten war die Idee dies Jahr erneut, einen Knoten zu erstellen der die autonomen Knoten steuern kann. Dabei kann der Kontrollknoten die autonomen Knoten starten und beenden.

2.2. Nachrichtenformat

Für die Definition des Nachrichtenformats wurde in diesem Jahr erneut auf Protocol Buffer zurückgegriffen. Dabei beinhaltet mein "Protokoll" diesesmal folgende Felder:

- sourceIP
- sourcePort
- sourceId
- NachrichtenTyp
- KontrollTyp
- NachrichtenInhalt
- zeitStempel

Dabei wird durch das setzen des Feldes *NachrichtenTyp* festgelegt, ob es sich um eine Anwendungsnachricht oder Kontrollnachricht handelt. Bei einer Kontrollnachricht sollte auch das Feld *KontrollTyp* gesetzt sein und die Werte *INITIALISIEREN* oder *BEENDEN* beinhalten.

Der *NachrichtenInhalt* wird primär für die Übertragung des Gerüchts genutzt.

Alle anderen Felder sollten selbsterklärend sein.

2.3. Erläuterung der Softwarestruktur

Von der Struktur her existiert zum einen eine Implementierung eines Kontrollknotens und zum anderen die eines unabhängigen Knotens.

Der Quellcode ist in Go in sogenannte Pakete unterteilt.

2.3.1. Client - Server

Bei der Implementierung habe ich mit der Überlegung begonnen, wie eine Datenstruktur geschaffen werden kann welche die geforderten Funktionalitäten bereitstellt. Zu diesem Zweck habe ich ein Client und Server "Objekt" erstellt, wobei das Server-Objekt die Funktionalitäten des Clients *implementiert* oder anders gesagt, von diesem *erbt*. Da in Go keine Klassen existieren, man trotzdem etwas ähnliches wie eine Vererbung oder die Deklation von Funktionen für einen Typ implementieren kann, tue ich mich bei dieser Benennung etwas schwer. Tatsache jedoch ist, dass genau dies bei meiner Implementierung der Fall ist. Ein Client verwaltet einen Namen, das verwendete Protokoll und eine IP Adresse. Der Server erweitert den Client um einen Port.

2.3.2. main()

Zum starten der Applikation wurde ein avaStarter angelegt, der mit verschiedenen Kommandozeilenparametern gesteuert werden kann. Dabei ist eine der wichtigsten Optionen *isController*. Mit dieser wird dem Programm gesagt welche Routine angestoßen werden soll. Entweder verhält sich das Programm wie ein Kontrollknoten, der von außen alles steuern kann, oder als unabhängiger Knoten.

2.3.3. Kontrollknoten

Der Kontrollknoten dient dabei hauptsächlich dazu, die in der Knotenliste angegebenen Knoten aufzulisten um es dem Benutzer zu ermöglichem, mit einem dieser Knoten auf Basis der Kontrollnachrichten zu kommunizieren. Dabei handelt es sich um die beiden Nachrichtenarten **INITIALISIEREN** und **BEENDEN**.

2.3.4. Unabhängige/Autonome Knoten

Diese beinhalten die eigentliche Logik dieser Übungsaufgabe. Sie kommunizieren miteinander sobald eine Nachricht eintrifft.

2.3.5. Generierung eines Graphviz Graphen

Im Ordner *generateGraphviz* befindet sich die ausführbare Quelldatei für die im Verzeichnis *graph* definierten Funktionen für das Erstellen beziehungsweise Generieren eines gerichteten oder ungerichteten Graphen. Zusätzlich zur Generierung eines Graphen kann mithilfe des *dot* Programmes der generierte Graph als JPEG exportiert werden.

2.4. Hinweise auf Implementierungsbesonderheiten

Die Besonderheiten sind zum einen die Implementierung der Tests, welche mir an manchen Stellen viel Zeit erspart haben indem Sie mich auf einen Fehler meiner Erweiterungen hingewiesen haben.

Außerdem wurden Goroutinen verwendet die mit hilfe von Channels miteinander Kommunizieren können.

Bei der Implementierung der Generierung von Graphen war es mir außerdem wichtig, dass ich sehe, was ich erstelle. Somit habe ich zuerst ein Shell-Skript geschrieben mit dem ich die Graphen als JPG, PNG, PDF oder SVG exportieren kann. Da ich jedoch nicht immer ein zusätzliches Shell-Skript aufrufen wollte, entschied ich mich dazu diese Funktion in mein Go Programm einzubauen.

Auch wenn es nichts so besonderes ist, bin ich doch ein wenig Stolz auf den verwendeten Tabwriter, der die Ausgabe der Kommandozeile in einer Art mit Tab seperierten Liste darstellt.

Eine negative Besonderheit die ich bis jetzt noch nicht beheben konnte ist, dass die Prozesse anscheinend zu schnell sind und ich dadurch beim senden einer Kontrollnachricht vom Kontrollknoten aus, je nach Situation, 5 Minuten warten muss bis die Nachricht bei dem jeweiligem Knoten ankommt. Das macht das Testen über längere Zeit sehr mühsehlig. Ich habe bereits versucht an verschiedenen stellen Sleeps einzubauen, welches sich jedoch nicht so ausgewirkt hat wie ich es mir erhofft hatte. Bei meinem letzten Versuch die Kommunikation auf UDP umzustellen musste ich jedoch feststellen, dass wie bereits im letzten Jahr, Pakete verloren gehen wenn der Empfangspuffer voll läuft. Mir ist bekannt, dass man dieses Puffer erhöhen kann, jedoch bringt das nur bis zu einem bestimmtem Punkt etwas.

2.5. typische Beispielabläufe

Ich habe im Ordner *exercise1* einen weiteren Ordner mit dem Namen *example* angelegt welcher zum einen Makefile enthält um die Quelldateien neu zu bauen. Außerdem beinhaltet dieses Verzeichnis mehrere Skripte. Zum einen um einen Kontrollknoten und zum anderen um autonome Knoten zu starten. Dafür werden außerdem die Dateien *Nodes.txt* und *Graphviz.txt* benötigt.

Man kann sich entweder dafür entscheiden die Knoten im *Gerücht-Modus* oder normal laufen zu lassen. Dafür startet man eine Shell und fürt das gewünschte Skript aus.

In einer weiteren Shell kann daraufhin das Skript für den Kontrollknoten aufgerufen werden. Mit diesem kann man sich daraufhin einen Knoten auswählen den man anstoßen möchte.

Die Ausgaben der Knoten werden in Textdateien umgelenkt und können beispielsweise nach dem beenden der Programme eingesehen werden.

Bei der Ausführung der autonomen Knoten im Gerüchte-Modus, schreibt jeder Knoten der das Gerücht glaubt eine Textdatei mit dem Gerücht in das aktuelle Verzeichnis. Dies erleichtert die Beurteilung, welcher Knoten das Gerücht geglaubt hat und wer nicht.

2.6. Fazit (gewonnene Erkenntnisse)

Aus dieser Übung habe ich hauptsächlich nur das erlernen der Sprache Go mitgenommen, da die Aufgabenstellung im letzten Jahr sehr ähnlich beziehungsweise teilweise identisch war.

Ein weiterer Teil meiner Erkenntnis bezieht sich auf etwas, was bereits im der Vorlesung *Softwareentwicklung für Kommunikationsnetze* bemerkt wurde. Dabei handelt es sich um die Tatsache, dass ein gutes Loggin bei solchen Anwendungen mit das wichtigste ist, da man sonst kaum weiß wo oben und unten ist.

Chapter 3. Exercise 2

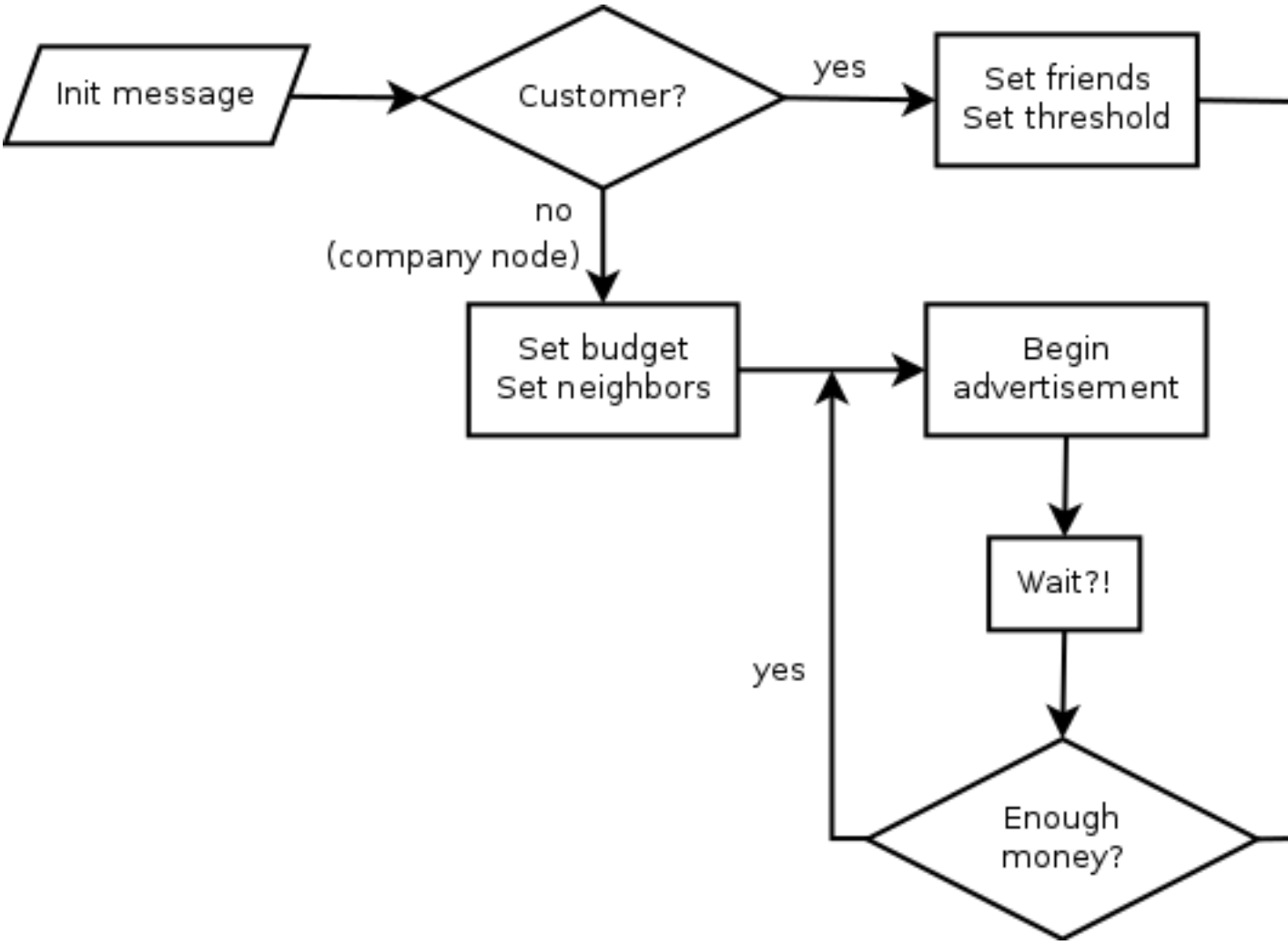
Warning

Diese Aufgabe ist so gut wie kaum implementiert. Es wurde alles vorbereitet, jedoch nicht zu Ende geführt.

3.1. Erläuterung der Idee

Die Idee bei dieser Aufgabe war es, sich den Ablauf zu zeichnen und anschließend einfach herunter zu programmieren.

Figure 3.1. Eingehende Initialisierung



In Bild 1 ist zu sehen wie ein Knoten reagieren soll wenn eine Init Nachricht eintrifft. Dabei wird geprüft ob es sich um einen Kunden handelt. Ist dies der Fall, so werden Freunde festgelegt und die Grenzwerte für das kaufen und weitererzählen von Produkten festgelegt.

Sollte es sich um ein Unternehmen handeln, so wird zuerst das Budget und die Nachbarn festelegt. Sobald dies geschehen ist kann mit der Werbung begonnen werden. Dies wird solange wiederholt, solange noch genügend Geld vorhanden ist.

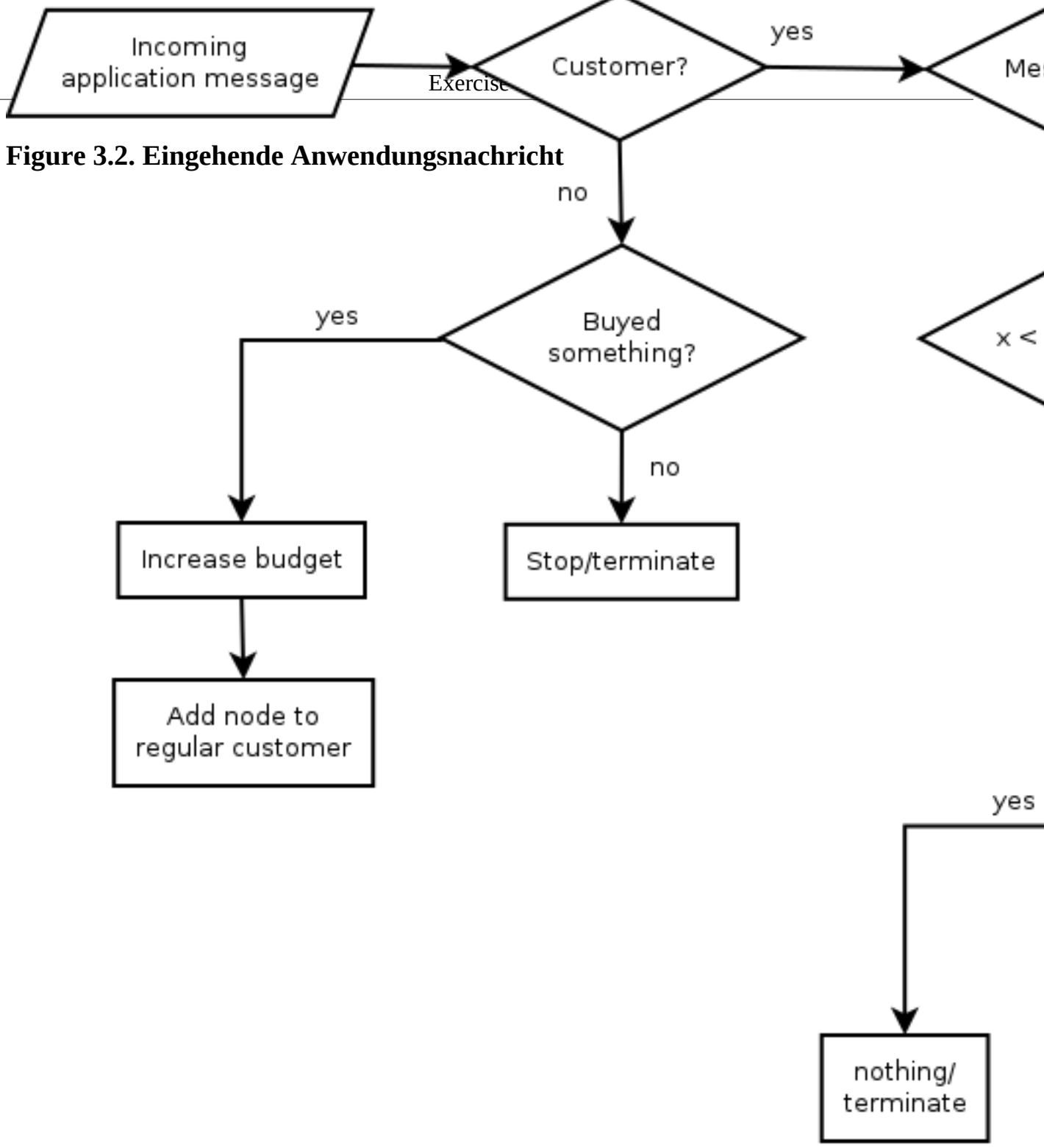


Figure 3.2. Eingehende Anwendungsnachricht

Bild 2 zeigt das Szenario einer einkommenden Anwendungsnachricht. Sollte es sich um einen Unternehmen handeln, so wird geprüft ob etwas gekauft wurde. Falls dies der Fall ist, wird das Budet erhöt und der Knoten als Kunde hinzugefügt, andernfalls wird der Vorgang beendet.

Der Kunde muss hingegen prüfen ob es sich um eine Info eines Freundes oder Unternehmens handelt. Darauf werden die jeweiligen Schwellwerte überprüft und im Falle eines nicht erreichten Wertes nachgeschaut, ob das Produkt bereits gekauft wurde. Falls der Kunde das Produkt noch nicht besitzt wird es gekauft und den Nachbarn mitgeteilt.

3.2. Nachrichtenformat

Das Nachrichtenformat ist identisch zu dem aus Aufgabe 1 mit dem einzigen Unterschied, dass ein Feld vom Typ *KnotenTyp* eingefügt wurde. Damit soll die Unterscheidung Kunde $<\!\!\rightarrow$ Unternehmen durchgeführt werden.

3.3. Erläuterung der Softwarestruktur

Die Struktur dieser Applikation ist ähnlich zu der aus Aufgabe 1. Es wurden jeweils nur noch zwei weitere Typen hinzugefügt um einen Kunden und ein Unternehmen representieren und verwalten zu können.

3.4. Hinweise auf Implementierungsbesonderheiten

Hier sind nur die Tests zu nennen, da sonst nichts besonderes realisiert wurde.

3.5. typische Beispielabläufe

Wenn es denn laufen würde, wäre das Prinzip analog zu Aufgabe 1 anzuwenden.

3.6. Fazit (gewonnene Erkenntnisse)

Das jährliche "In den Weihnachtsferien werde ich schon genügend Zeit haben." erneut gekommt wiederlegt.

Chapter 4. Exercise 3

Warning

Nur zu ca 90-95% umgesetzt, da noch ein Problem auftritt.

4.1. Erläuterung der Idee

Die Idee bei Aufgabe 3 war, eine sequenzielle Anwendung zu erstellen um daraufhin einen weiteren Prozess hinzu zu nehmen um dann die daraus resultierenden Probleme zu beheben.

Dabei wurde ein sogenannter **FileManager** und **FileUser** erstellt. Der FileManager verwaltet eine Datei und kann Berechtigungen an Benutzer vergeben. Das schreiben in die Datei wurde von mir so implementiert, dass der Benutzer den Pfad zur Datei ausgehändigt bekommt (Aufgrund der Anforderung, dass es nur auf localhost lauffähig sein soll) und diese daraufhin selbst beschreiben darf.

4.2. Nachrichtenformat

Für diese Aufgabe wurde ein abgewandeltes Nachrichtenformat benutzt. Es wurden drei Nachrichten definiert: * FilemanagerRequest * FilemanagerResponse * GoldmanToken

Der FilemanagerRequest wird von einem Benutzer an den Verwalter gesendet der wiederum mit einem FilemanagerResponse antwortet. Somit empfängt ein Manager immer Requests und ein Benutzer immer Responses.

Sollte es zu einem vermutlichem Deadlock kommen (was bei nur 2 Managern immer der Fall ist), wird das GoldmanToken zu dem Prozess gesendet der als blockierender Prozess in der Response angegeben wurde.

4.3. Erläuterung der Softwarestruktur

Von der Struktur her existieren diesmal zwei Quellcode Dateien. Eine für den Verwalter und eine für den Benutzer.

Die verwendete Datenstruktur ist weiterhin das Server-Object welches bereits für Aufgabe 1 verwendet wurde.

4.4. Hinweise auf Implementierungsbesonderheiten

Es wurden, wie auch schon in Aufgabe 1 und 2, Goroutinen eingesetzt. Diesmal wurde jedoch kein Channel verwendet, sondern ein Mutex, der sicherstellen soll, dass die ID desjenigen der die Datei derzeit belegt korrekt ist.

Auch wurde eine sogenannte Closure genutzt um eine Variable als Funktion zu definieren um je nachdem ob es sich um einen Prozess mit gerader oder ungerader ID handelt eine andere Funktion

zuweist. So konnte in der "Hauptschleife" nur diese eine Funktion verwendet werden ohne das immer unterschieden werden musste ob es ein gerader oder ungerader Prozess ist.

Das wichtigste ist jedoch, dass die Benutzer jeweils nur mit geraden Protnummern ausgestattet werden dürfen. Dies wurde aus dem Grund definiert, weil der darauffolgende ungerade Port dafür verwendet wird um das Token zu empfangen. Bei dem Empfangen unterschiedlicher Nachrichtenarten auf einem Port kam es bei der Verwendung von Protobuf zu problemen, da dieser nicht genau feststellen konnte um welchen Nachrichtentyp es sich handelt.

4.5. typische Beispielabläufe

Wie auch bei den anderen Aufgaben existiert im *exercise3* Ordner ein Verzeichnis namens *example* in dem ein Makefile und Shell Skripte zu finden sind.

Die Datei *startSkript.sh* wurde nicht oft verwendet. Der Ablauf der meist genutzt wurde war, dass vier Terminals gestartet wurden. In den ersten beiden wurden ein ManagerA und ein ManagerB, zu denen jeweils auch Start-Skripte existieren, gestartet. Die zwei verbleibenden Terminals werden jeweils durch das Skirpt *startFileUser.sh* in Betrieb genommen. Gibt man diesem Skript keinen Parameter mit, so wird eine Version des FileUsers gestartet. Wird mindestens ein Parameter mitgegeben, so führt das Skript einen FileUser mit anderen Einstellungen aus.

4.6. Fazit (gewonnene Erkenntnisse)

Bei dieser Aufgabe habe ich wiedereinmal feststellen müssen, dass Aufgaben manchmal leichter klingen als sie eigentlich sind. Ich bin davon ausgegangen, dass ich diese Aufgabe in kürzester Zeit lösen könne, hatte jedoch so viele Probleme beim Debuggen und beim finden von Fehlern.

Es war desweiteren Ärgerlich nach geraumer Zeit feststellen zu müssen, dass unterschiedliche Nachrichten auf dem selbem Port schwierig zu handhaben sind.

Ansonsten war dies eine sehr spannende Aufgabe, die mir (und teilweise meinen Kollegen bei der Consistec) sehr viel Spaß gemacht hat.

Chapter 5. Fazit (Übergreifend)

Zusätzlich zu den Fazits der einzelnen Übungen wollte ich noch ein übergreifendes Fazit abgeben. Dabei handelt es sich vorallem darum, dass Dinge, die man sich vornimmt, nicht immer eingehalten werden. Wenn man bedenkt, dass ich auf gut Glück bereits gegen ende August mit der Bearbeitung der im letzten Jahr erschienenen Aufgabe 1 begonnen habe, so bin ich wirklich nicht weit gekommen. Zwar war ich mit der ersten Aufgabe zeitnah (für das Semester) fertig, jedoch hat das dazwischenliegende Praktikum einen größeren Strich durch die Rechnung gemacht, als ich zu beginn von ausgegangen bin.

Desweiteren muss ich eingestehen, dass Go einige merkwürdige beziehungsweise gewöhnungsbedürftige Konzepte besitzt mit denen ich mich recht schwer getan habe. Dafür wurde mir jedoch das Leben durch die Verwendung von GitHub etwas erleichtert. Ich nutzte die dort angebotene Funktion *Issues* einzureichen welche ich ab und an erstellte um diese nach und nach abzuarbeiten. Außerdem vereinfacht die *git gui* die Arbeit beim comitten, so dass nur bestimmte teile aus einer Datei heraus comitted werden.